Build android apk from command line without gradle

2023-07-19

Introduction
Building regular android app
Adding JNI support
Conclusions

Introduction

This is post is similar to the one about building flutter but a bit different. Actually this one was written a few years prior to writing flutter version but just never got published. Since then some steps has changed and I moved to Java 11. But you if you still build with java 8 old build method is still surprisingly functional.

So the idea is to build and APK file which then can be installed on an android phone without using gradle and doing each step “manually”. If you are building regular android apps with Java/Kotlin then this is pointless. But it has a bit more sense when you are building game like apps with C/C++. But the most important thing is just to learn how this build process is working, what are its parts and how they stack together.

Here are the files that will be used:

MainActivity.java
AndroidManifest.xml
activity_main.xml
styles.xml
android_handmade.cpp

Building regular android app

Let’s start with building very small android app without any help of build tool. For the initial android build we need only 2 files: **MainActivity.cpp, AndroidManifest.xml**. These should be enough to build the apk. We will need one more file **keystore.jks** to build to sign the apk so that android will allow us to install the apk on the device.

So first of all let’s create a root project folder. Name it whatever you want. I will name it ‘handmade_native_android’ and then cd into it. Inside that folder create these files:

```
./AndroidManifest.xml
./........./MainActivity.java
```

We put **MainActivity.java** in **/java/com/hereket/handmade_native_android/** folder structure. **java** is the place where we put all our java code and **com/hereket/handmade_native_android/** just reflects our package name ‘com.hereket.handmade_native_android’.

These two should be enough to be and apk if we create UI programmatically. But I want to add some resource files so that if you want to create UI as usual or use resource files, you won’t have to figure out it yourself.

So for that let’s add a little more files and folder. In the root folder of the project add a folder named **‘res’**. Here we will put required android resources. Inside it lets create two more folders **layout** and **values**. And then two more files **layout/activity_main.xml** and **values/styles.xml**.

My android SDK is installed here “**/opt/Android/Sdk**”. So for you own tests replace this path with the pass of android installation on your machine. Also in order to use android tools I added path the binaries to the path like this:

```
export PATH="/opt/Android/Sdk/build-tools/33.0.2:$PATH"
```

We will need one more folder **__build**. Put it in the root folder of the project. It is a temporary folder which will contain all temporary files needed for the compilation process.

So let’s start generating intermediate files. For that run the following command. It will create **R.java**. It will contain all the resources with their respective ids which will be used in our activity. The file will be created in in __build/gen/…

```
aapt package -f -m \
    -J __build/gen \
    -S res \
    -M AndroidManifest.xml \
    -I "/opt/Android/Sdk/platforms/android-33/android.jar"
```

The next we just compile MainActivity.java and R.java into their object form.

```
javac \
    -classpath "/opt/Android/Sdk/platforms/android-33/android.jar" \
    -d "__build/obj" \
    "__build/gen/com/hereket/handmade_native_android/R.java" \
    java/com/hereket/handmade_native_android/MainActivity.java
```

This command will create MainActivity.class, R.class and class files for inner classes of R.java. They will be placed in “__build/obj/…”. Now create ‘apk’ folder inside ‘__build’ folder for the next step. In the next step let’s dex all class files and merge all our class files into a my_classes.jar file.

```
d8 __build/obj/**/*.class \
    --output __build/apk/my_classes.jar \
    --no-desugaring \
```

The “**/*.class” is expanded int a list of class files. If you have issues with this method just use **find** utility to find a list of all class file, put it into a varible and pass it instead. And now let’s merge this and android.jar and create **classes.dex** file. It is a merged ‘dex’ format file that can run on dalvik.

```
pushd __build/apk
d8 \
    "/opt/Android/Sdk/platforms/android-33/android.jar"
    my_classes.jar
popd
```

So we got all the required files let's merge them into an apk. For that we again will use **aapt**:

```
aapt package -f -M AndroidManifest.xml -S res \
    -I "/opt/Android/Sdk/platforms/android-33/android.jar" \
    -F __build/handmade_native_android.unsigned.apk \
    __build/apk/
```

Yay. We got our an apk file named **handmade_native_android.unsigned.apk** inside __build/apk/ folder. But we cannot run it on our device because we need a couple more steps in order to prepare it to to format that android actually accepts. The next step is to align apk and we can do with this command.

```
zipalign -f -p 4 \
    __build/handmade_native_android.unsigned.apk\
    __build/handmade_native_android.aligned.apk
```

This will create a file name **handmade_native_android.aligned.apk** inside __build/apk. The last step is to sign the apk with our key. It is a required and android security system will not accept an apk without beign signed first. If you have your key you can just drop into the root folder with the name 'keystore.jks' of this project or you just want to follow along just run this:

```
keytool -genkeypair -keystore keystore.jks -alias androidkey \
      -dname "CN=hereket.com, OU=ID, O=HEREKET, L=Abc, S=Xyz, C=GB" \
      -validity 10000 -keyalg RSA -keysize 2048 \
      -storepass android -keypass android
```

So after you copied key or generated key we can need to sign the apk.

```
apksigner sign --ks keystore.jks \
    --ks-key-alias androidkey \
    --ks-pass pass:android \
    --key-pass pass:android \
    --out __build/handmade_native_android.apk \
    __build/handmade_native_android.aligned.apk
```

And here created a file **handmade_native_android.apk**. This is a final apk that could be run a device.

Adding JNI support

We have achieve our goal of creating a runnable apk but let's add a little bit of JNI support. It is not required.

For this to work let's first and some code. Inside the root folder add **'jni'** folder and inside of it add a file named **android_handmade.cpp** .

In order for the JNI actually do something that we can see let's change our MainActivity.java file slightly to actually call JNI code. You can see the new code here: **New MainActivity.java**

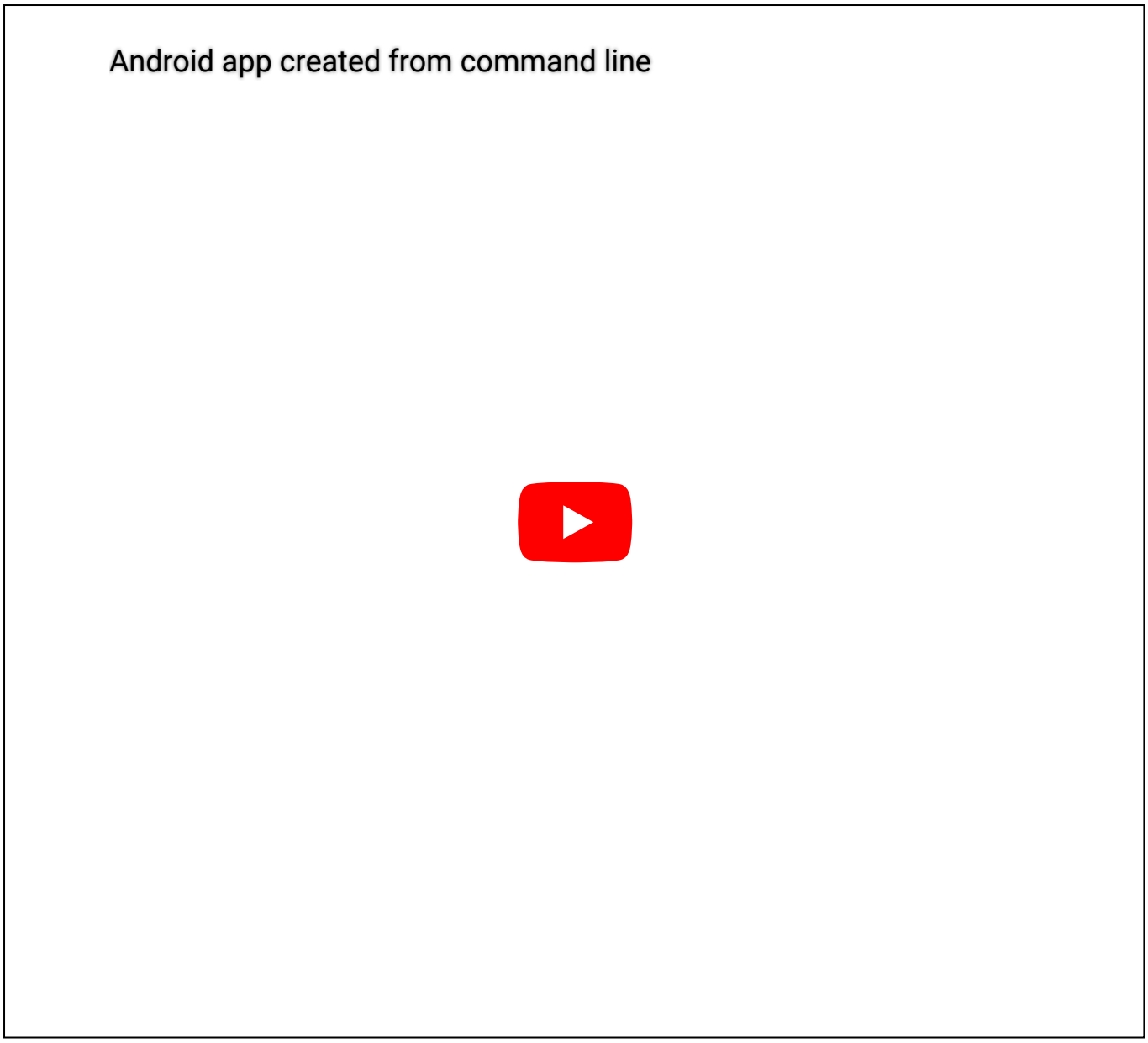
```
ARMV7="/opt/Android/Sdk/ndk/r25b/toolchains/llvm/prebuilt/linux-x86_64/bin/armv7a-linux
MY_LOCAL_LDFLAGS='-ljnigraphics -llog -landroid'
CFLAGS="-g -O0 -fPIC -shared -static-libstdc++"

"${ARMV7}" $CFLAGS \
    -o __build/apk/lib/armeabi-v7a/libandroid_handmade.so \
    jni/android_handmade.cpp \
    $MY_LOCAL_LDFLAGS
```

Here we just create a shared library named **libandroid_handmade.so** inside '__build/apk/lib/armeabi-v7a'. In reality I should have create a separate shared library for 64 bit arm and maybe a version for x86 but since I am targetting just android and 64 bit arm can execute 32 bit code I can just skip those steps.

So now we have to redo all the steps above starting from the place where are starting to create apk, align and sign it. After that we should have an apk that is using C/C++ code.

As a result we will get this app.



Conclusions

In this article we created a fully functional apk that runs on a real device. The only reason to do so was to look at pieces used to create an apk and the processes used to put those pieces together.

I do not recommend building **real** android apps using this way but you should try to do this process manually once just to get the grasp of what is happening in the build process.

If you want to have a look at the full final project then go to the github page at **https://github.com/hereket/handmade_native_android**.

© 2024, Hereket RSS feed